CS 4414: Recitation 5

Sagar Jha

Today's agenda

Compiler optimizations (BO Chapter 5)

- What is the goal of optimization?
- Tricky questions
- Compilation techniques: Code motion, Out-of-order execution, Data flow analysis, Loop unrolling, Inline expansion
- g++ optimization options
- C++ specific optimizations

Iterators and Algorithm (BS Chapter 12)

- What are iterators? How to use them?
- Type of iterators
- Overview of algorithms

What is the goal of optimization?

What is the goal of optimization?

- Improve program performance *without changing its behavior*
- C++ compilers must follow the as-if-rule: All optimizing transformations are allowed as long as they do not change the "observable behavior" of the program
- Notable exceptions:
  - Undefined behavior
  - Copy elision
  - Return value optimization

Tricky question 1

- Are the following programs equivalent?

```
void twiddle (long *xp, long *yp) {
   *xp += *yp;
   *xp += *yp;
}

void twiddle (long *xp, long *yp) {
   *xp += 2 * *yp;
}
```

Tricky question 2

- Are the following programs equivalent?

```
long f();

long func1() {
  return f() + f() + f() + f();
}

long func2() {
  return 4*f();
}
```

Loop-invariant code motion

```
length (my_vector v);

for (int i = 0; i < length(v); ++i) {
  // access v[i]
}

int len = length(v);
for (int i = 0; i < len; ++i) {
  // access v[i]
}
```

Loop-invariant code motion

```
void lower1(char *s) {
  for (int i = 0; i < std::strlen(s); ++i) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}

void lower2(char *s) {
  long len = strlen(s);
  for (int i = 0; i < len; ++i) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}
```

Out-of-order execution

- Modern processors can execute multiple instructions in parallel
- The degree of parallelism depends on how independent individual instructions are
- Reorder instructions based on availability of input data and execution unit
- A form of data-flow analysis/computation

Data flow analysis

- Compute possible values of variables at different points in the program during compilation

```
if (some_bool) {
  x = 1;
} else {
  x = 3;
}

if (x < 10) {
  // do something
}
```

Loop unrolling

- Reduces the number of iterations for a loop

```
int prod = 1;
for (int i = 0; i < length; i++) {
  prod *= data[i];
}

int prod = 1;
for (int i = 0; i < length; i+=2) {
  prod *= data[i] * data*[i+1];
}
// one more step if data has odd number of elements...
```

Loop unrolling

- Using multiple accumulators can improve performance

```
for (int i = 0; i < length; i+=2) {
  prod_even *= data[i];
  prod_odd *= data[i+1];
}
```

Function inlining and consts

- Inline expansion, by placing a copy of the function at call site, can remove function-calling overheads
- C++ offers the *inline* keyword to suggest inlining to the compiler, in most cases, you don't need to manually specify it
- Const, likewise, is for improving program readability and correctness
- Compilers can often figure out const-related optimizations by themselves

Branch prediction

- Branches (if-else conditions, loops) interfere with instruction pipelining
- Branch prediction tries to prefetch instructions by betting on the result of the condition, backtracking if needed
- Most upvoted stackoverflow question: https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array
  Performance of processing a sorted array is almost six times faster
  Summary: predicting data[c] > 128 in the user's code is almost always successful with a sorted array

Aggressive optimization can potentially reduce performance!

- Aggressive inlining and loop unrolling can increase code size
- Larger instruction size reduces the performance of the instruction cache
- g++ optimization levels:
  - -O0: default, no optimizations – useful for debugging
  - -O1: core optimizations (function inlining, tail recursion, not calling functions with no side-effects, reusing stack space of variables no longer used) – decent debugging experience
  - -O2: more aggressive inlining and loop unrolling, vector instructions for simple loops and independent operations – industry standard
  - -O3: even more aggressive inlining and unrolling – impossible to debug
  - -Oz: smallest possible code size, useful when executing on microprocessors

Live demo on https://godbolt.org

Other C++-specific optimizations

- RAII for predictable performance (and not garbage collection)
- Garbage collection (in Java etc.) may be potentially inefficient:
  - Unpredictable performance: The program may be paused for garbage collection to run, if the program is running out of memory
  - Heavy RAM usage: program uses more memory because objects are not cleaned up right when they go out of scope
  - Memory leaks possible in some cases
  - Scalability: Garbage collection performance may be worse with small number of threads
- Copy elision: Eliminate unnecessary copying of objects. E.g. not copying a temporary class object into another object
  - Return value optimization (RVO): Eliminate temporary object holding a function’s return value

RVO can change program behavior!

```
#include <iostream>

struct C {
  C() = default;
  C(const C&) { std::cout << "A copy was made.\n"; }
};

C f() {
  return C();
}

int main() {
  std::cout << "Hello World!\n";
  C obj = f();
}
```

```
Hello World!
A copy was made.
A copy was made.
```

```
Hello World!
A copy was made.
```

```
Hello World!
```

What does compiler optimization mean for programmers?

- Classic dilemma: Abstraction vs. performance
- Develop good coding habits informed with program performance characteristics
- Profile code with gprof to gain insights into program's performance. Implement optimizations accordingly – performance bottleneck analysis (HW2)
- Do not prematurely optimize and complicate code-logic without understanding the impact
  "Premature optimization is the source of all evil" – Donald Knuth

What does compiler optimization mean for programmers?

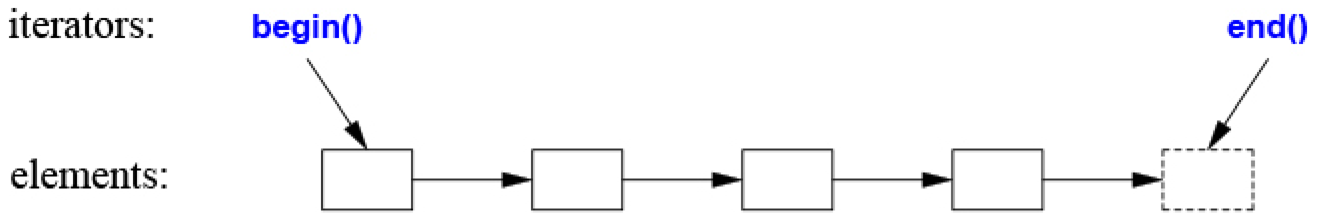
```diff
 bignum::Bignum bignum::Bignum::operator*(const Bignum& other) const {
     Bignum prod(num_digits() + other.num_digits());

-    const Bignum& smaller = (*this < other ? *this : other);
-    const Bignum& larger = (*this < other ? other : *this);
+    // const Bignum& smaller = (*this < other ? *this : other);
+    // const Bignum& larger = (*this < other ? other : *this);

-    for(uint32_t i = 0; i < smaller.num_digits(); ++i) {
+    std::reference_wrapper<const Bignum> smaller = other;
+    std::reference_wrapper<const Bignum> larger = *this;
+
+    if (*this < other) {
+        smaller = *this;
+        larger = other;
+    }
+
+    for(uint32_t i = 0; i < smaller.get().num_digits(); ++i) {
         uint32_t carry = 0;
-        for(uint32_t j = 0; j < larger.num_digits(); ++j) {
-            prod[i + j] += smaller[i] * larger[j] + carry;
+        for(uint32_t j = 0; j < larger.get().num_digits(); ++j) {
+            prod[i + j] += smaller.get()[i] * larger.get()[j] + carry;
             carry = prod[i + j] / 10;
             prod[i + j] %= 10;
```

What is an iterator?

- Used for iterating through a container.
  Why not use a for(int i = 0; i < container.size(); ++i) loop?
- *Abstracts* the container and provides access to elements. *Separates* the algorithm from the container.
  For example, sort(container.begin(), container.end()); can sort a vector or a list
- Special iterators: begin(), end(), rbegin(), rend()



Iterators: Use cases – std::sort

```
void f(vector<Entry>& vec, list<Entry>& lst)
{
     sort(vec.begin(),vec.end());                           // use <
for order
     unique_copy(vec.begin(),vec.end(),lst.begin());   // don't
copy adjacent equal elements
}
```

```
bool operator<(const Entry& x, const Entry& y)   // less than
{
     return x.name<y.name;       // order Entries by their names
}
```

```
list<Entry> f(vector<Entry>& vec)
{
     list<Entry> res;
     sort(vec.begin(),vec.end());
     unique_copy(vec.begin(),vec.end(),back_inserter(res));
// append to res
     return res;
}
```

Iterators: Use cases – std::find

```
bool has_c(const string& s, char c)     // does s contain the
character c?
{
     return find(s.begin(),s.end(),c)!=s.end();
}
```

```
template<typename T>
using Iterator = typename T::iterator;         // T's iterator

template<typename C, typename V>
vector<Iterator<C>> find_all(C& c, V v)        // find all
occurrences of v in c
{
     vector<Iterator<C>> res;
     for (auto p = c.begin(); p!=c.end(); ++p)
           if (*p==v)
                 res.push_back(p);
     return res;
}
```

Type of iterators

- Iterators provide a ++ operator to point to the next element, * for directly accessing the element
- A vector iterator may be different from a list iterators
- Stream Iterators
  - Input/output iterators

```
ostream_iterator<string> oo {cout};    // write strings to cout

int main()
{
    *oo = "Hello, ";    // meaning cout<<"Hello, "
    ++oo;
    *oo = "world!\n";   // meaning cout<<"world!\n"
}
```

  - std::stringstream, std::ifstream, std::ofstream

Predicates

- A function that returns true or false
- Can pass to some algorithm that uses iterators to filter the results

```
auto p = find_if(m.begin(), m.end(), [](const auto& r) { return
r.second>42; });
```

Overview of algorithm

- for_each – run a function for each element in a container
- find – find the first match
- count – count the number of occurrences
- replace, replace_if – Replace elements selectively
- copy, move, merge – copy/move/merge containers
- binary_search – search for an element in a sorted container (logarithmic for RandomAccessIterators, linear otherwise)
- transform, generate, fill, rotate, max, min…